Stratified Tree Search: A Novel Suboptimal Heuristic Search Algorithm

Levi H. S. Lelis
Dept. of Computing Science
University of Alberta
Edmonton, Canada
levi.lelis@ualberta.ca

Sandra Zilles
Dept. of Computer Science
University of Regina
Regina, Canada
zilles@cs.uregina.ca

Robert C. Holte
Dept. of Computing Science
University of Alberta
Edmonton, Canada
rholte@ualberta.ca

ABSTRACT

Traditional heuristic search algorithms use the ranking of states that a heuristic function provides to guide the search. In this paper—with the objective of improving suboptimality and runtime of search algorithms when only weak heuristics are available—we present Stratified Tree Search (STS), a suboptimal heuristic search algorithm that uses a heuristic to partition the state space to guide the search. We call this partition a type system. STS assumes that nodes of the same type will lead to solutions of the same cost. Thus, STS expands only one node of each type in every level of search. We show that in general STS offers a good tradeoff between solution quality and search speed by varying the size of the type system. However, in some cases, STS might not provide a fine adjustment of this tradeoff. We present a variant of STS, Beam STS (BSTS), that allows one to make fine adjustments of this tradeoff. BSTS combines the ideas of STS with those of Beam Search. Our empirical results in benchmark domains show that both STS and BSTS can find solutions of lower suboptimality in less time than standard heuristic search algorithms for finding suboptimal solutions.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms

Keywords

Heuristic Search; Suboptimal Search; Planning; Stratified Sampling

1. INTRODUCTION

Heuristic search is the building block for various algorithms designed for autonomous agents – see for instance, Koenig and Likhachev [11] and Yeoh et al. [20]. Heuristic search algorithms use a heuristic function to guide their search to find solutions for state-space problems. A heuristic function $h(\cdot)$ estimates the cost-to-go of a node, i.e., it estimates the optimal cost of a solution path from a given node. Intuitively, nodes with lower heuristic value are more promising than nodes with higher heuristic value as they seem to be closer to a solution. Heuristic search algorithms such as A* [6] and IDA* [12] use the function $f(s) = g(s) + h(s)$ to guide their search, where $g(s)$ is the cost from a start state s^* to node s in the search tree. $f(s)$ is an estimate of the lowest cost of all solutions for s^* that go through s. A* and IDA* usually explore fewer nodes while searching for a solution if $h(\cdot)$ offers accurate estimates of the cost-to-go.

Beam search algorithms such as BULB [5] use heuristic functions in a different way to guide their search. BULB selects for expansion the B nodes with the lowest f-value at a given level of search. Thus, in this case, the accuracy of the heuristic estimates is no longer important. For BULB, ideally, a heuristic function will correctly rank nodes in the search tree: nodes closer to a goal receive lower heuristic value than nodes farther from a goal. In contrast with A* and IDA*, here the absolute value is not important. The existence of different strategies for using a heuristic function allows one to choose the appropriate algorithm for existing heuristic functions in practical scenarios.

In this paper we present a novel heuristic search algorithm that uses a heuristic function in yet a different way. Our goal was to develop a general-purpose heuristic search algorithm for quickly finding near-optimal solutions in relatively large domains when only weak heuristics were available. When we refer to weak heuristics we mean heuristics that (1) do not provide accurate estimates of the cost-to-go and (2) do not provide an accurate ranking of the nodes in the search tree. Analogously, when we refer to strong heuristics we mean heuristics that either (1) provide accurate estimates of the cost-to-go or (2) provide an accurate ranking of the nodes in the search tree. While other heuristic search algorithms rely on the estimates of the cost-to-go or on the ranking provided by a heuristic for the nodes in the search tree, our algorithm uses a heuristic to group together nodes with similar solution cost. Consider the following example. Suppose an agent wants to find the shortest path from its current location to the closest grocery store. If it is known that heading south or west from its current location will lead to paths of the same length, then only one of the two initial directions (south or west) must be further explored for planning, the other direction can be ignored. Note that we do not need to know the estimated solution length by heading south or west nor which direction looks more promising—we only need to know which directions lead to solutions of the same cost.

Our algorithm, Stratified Tree Search (STS), is based on

Appears in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013), Ito, Jonker, Gini, and Shehory (eds.), May, 6–10, 2013, Saint Paul, Minnesota, USA.*

the Stratified Sampling method for efficiently estimating the size of backtrack search trees [2, 3]. We call the grouping system used by STS a *type system*; our method assumes that nodes of the same type lead to solutions of the same cost. Our algorithm resembles beam search in that it expands nodes in a level by level manner and in that it ignores a set of the nodes at every level of search. However, instead of expanding the B best nodes according to the heuristic, STS expands one node of each type at every level of search. STS's search is more diverse than beam search, in the following sense. If nodes of different types tend to have different heuristic values (the extreme case of this happens when the type of a node is defined as its heuristic value), then STS and beam search will have almost opposite behaviors: all the nodes expanded by beam search will have similar heuristic values whereas the nodes expanded by STS will tend to all have different heuristic values. STS finds suboptimal solutions because in practice nodes of the same type will not have identical solution costs. STS is fundamentally different than abstraction-based methods such as pattern databases (PDB) [4]. This is because the type system used by STS is only a partition of the state space – one cannot search in the type system space as there are no edges connecting types. A PDB, for instance, is constructed by searching backwards in the abstracted state space.

We compare STS to BULB and Weighted IDA* (WIDA*) [13], two standard suboptimal heuristic search algorithms that also scale to state spaces as large as the ones we use as testbeds, and also to IDA*-BST, which is IDA* using the inadmissible heuristic produced by bootstrap learning [10]. The Bootstrap system learns strong heuristics from initial weak heuristics. Our experimental results suggest that heuristic functions that are considered weak for traditional heuristic search algorithms can be used to quickly guide STS to near-optimal solutions. Our experimental results show that STS can (a) quickly find near-optimal solutions in the domains tested; (b) produce solutions of much lower suboptimality in much less time than WIDA*, BULB, and IDA*-BST. However, as we discuss later in the paper, we do not expect STS to perform better than other heuristic search algorithms when strong heuristics are available.

We show in this paper that STS offers a good tradeoff between solution quality and search speed by varying the size of the type system being employed, i.e., the number of types at every level of search. However, in some cases, STS does not provide a fine adjustment of this tradeoff. We present a simple variant of STS, named Beam STS (BSTS), that allows one to make fine adjustments of the tradeoff between solution quality and search speed. BSTS combines the ideas of Stratified Tree Search with those of Beam Search. BSTS expands the nodes that represent the "best" B types at every level of search. Our empirical results show that BSTS can, in some cases, find solutions of higher quality and in less time than STS. The empirical results also point out that BSTS can substantially outperform traditional heuristic search algorithms on the domains tested when weak heuristics are employed. Similar to STS, we do not expect BSTS to perform better than other heuristic search algorithms when strong heuristics are available.

2. STRATIFIED TREE SEARCH

The STS algorithm presented in this paper is based on a prediction method by Chen [2, 3], which we call Stratified Sampling (SS). SS uses a partition of the state space, which Chen called a stratifier but we will call a type system, to efficiently estimate the size of backtrack search trees. Chen assumed that nodes of the same type at a level of the search tree would root subtrees of the same size. Therefore, by sampling only one node of each type SS efficiently estimates the size of a search tree.

Chen also showed that SS can be used to measure any property of backtrack search trees [2]. Therefore, in theory, SS is applicable for finding suboptimal solutions for the least-cost path planning problem. However, Chen assumed trees with bounded depth, and applied SS in domains with high solution density. For instance, Chen applied SS for finding an approximation of the longest path for the uncrossed knight's tour problem. In the uncrossed knight's tour problem a knight is placed on a chess board and the goal is to find the longest tour the knight can make, with its chess-like moves (L-shaped), without crossing cells already visited. It is easy to obtain some solution in this domain as every leaf node (i.e., nodes not generating children) is a solution.

STS is a variation of SS that is able to find suboptimal solutions in combinatorial domains with unbounded depth that are much larger than the domains Chen used as testbeds. In experiments on the 35-pancake puzzle, 20-blocks world, and 5x5 sliding-tile puzzle (24-puzzle) SS did not find any solution on 10 random instances of each domain with a three-hour time limit per instance; STS finds near-optimal solutions in seconds in all three domains for all instances tested. STS works better than SS in these larger domains with lower solution density because it uses a heuristic function to define the type system that guides its search. A type system is defined as follows.

Algorithm 1 Stratified Tree Search

```
1: input: start state s, goal state g, and type system T
2: output: path from s to g
3: C ← {⟨s, 1⟩}; i ← 0
4: while true do
5:    empty N
6:    C ← expansion(C, i)
7:    i ← i + 1;
8:    if C is empty then
9:       return failure
10:   end if
11: end while
```

DEFINITION 1. *Let $S(s^*)$ be the set of nodes in the search tree rooted at s^*. $T = \{t_1, \ldots, t_n\}$ is a type system for $S(s^*)$ if it is a disjoint partitioning of $S(s^*)$. For every $s \in S(s^*)$, $T(s)$ denotes the unique $t \in T$ with $s \in t$.*

STS receives as input a start state s, a goal state g, and a type system T, and it returns a path from s to g. STS assumes that nodes of the same type will lead to solutions of the same cost. Under this assumption, exploring only one node of each type is enough to find the optimal solution. Algorithms 1 and 2 show the pseudocode. In Algorithm 2 STS keeps only two frontiers in memory, one for the current level of search (C), and another one for the next level of search (N). However, in our implementation of STS we keep all the levels in memory so that the path from start to goal can be recovered. STS stores pairs $\langle s, w \rangle$ in the frontiers, where s is a node at a given level of search such that $T(s) =$

Algorithm 2 expansion

```
1: input: layer C and cost i
2: output: layer N or path from s to g
3: for each element ⟨s, w⟩ in C do
4:   for each child ŝ of s do
5:     if ŝ equals g then
6:       return path with cost i + 1
7:     end if
8:     if N contains an element ⟨s', w'⟩ with T(s') = T(ŝ)
       then
9:       w'' ← w' + w
10:      with probability w/w'', replace ⟨s', w'⟩ in N by
         ⟨ŝ, w''⟩
11:    else
12:      insert new element ⟨ŝ, w⟩ in N.
13:    end if
14:  end for
15: end for
16: return N
```

t, and w is a weight associated with s. We call such a pair the *representative pair* of type t, as STS keeps at most one such pair for every $t \in T$ in each level. The value of w in a representative pair for type t is the estimated number of nodes of type t that exist at that level of search. Such weights were used by SS to estimate the size of a search tree. In SS, the sum of all w values gives an estimate of the size of a search tree. We use w to sample the search tree more uniformly.

Algorithm 1 starts by inserting the start state with a weight of one into the current frontier C. As in breadth-first search, STS iterates through an entire level before moving to the next level of search. In the current implementation of STS we assume edges with unit cost. STS generalizes to domains with non-unit edge costs if instead of iterating over levels STS iterates over layers of nodes of same g-cost. In Algorithm 2, STS selects for expansion a state s in a representative pair $\langle s, w\rangle$ in C. If the goal is found among the children of s, then STS returns the path through s with cost $i+1$. Otherwise, each child $\hat{s}$ of s is considered for insertion in N. When STS generates a node $\hat{s}$ of type t and there is no representative pair for t in N, then $\hat{s}$ is inserted with the weight of its parent in N. If there is a representative pair $\langle s', w'\rangle$ with $T(\hat{s}) = T(s') = t$ in N, then STS adds w to w' — STS estimated there were w' nodes of type t in N, and now w nodes in C generate a node of type t in N. STS replaces $\langle s', w'\rangle$ with the new representative pair $\langle \hat{s}, w'+w\rangle$ with probability $w/(w'+w)$. After expanding all nodes in representative pairs at a level of search STS starts expanding the next level of search.

Note that every node seen during search may potentially become the node in a representative pair. Nodes that were generated from parents with higher values of w have a higher chance of becoming a representative in exact proportion to their frequency of occurrence. STS samples the search tree more uniformly by weighting nodes in this way.

Multiple Probes – The process just described represents one probe of STS. A probe finishes when STS either finds a goal or reaches leaf nodes. In the latter case, C will be empty and the algorithm will return failure. One could run multiple probes and possibly improve the performance of STS by returning the solution with minimum cost found. An experiment illustrating the tradeoff between solution cost and runtime is discussed later (see Table 1).

Transposition Detection – The only kind of transposition detection we implemented in STS was parent-pruning, i.e., STS does not generate node $\hat{s}$ from s if $\hat{s}$ is the parent of s in the search tree. Methods for detecting longer cycles or transpositions could be implemented in STS, but we did not feel they would be worthwhile. The type systems we use substantially compress the state spaces (many nodes at each level of search are mapped to the same type). Therefore, STS already expands very few nodes during search, and the chance of encountering a large number of transpositions among the nodes it expands is rather low. Nevertheless, as with IDA*, memory-based methods for detecting transpositions could be implemented with STS. Also, other transposition detection methods such as the ones based on finite state machines [1, 19] can be easily implemented with STS.

Restarts – In some cases STS might be misled by the type system to dead-end parts of the state-space. In this case a probe will never finish. One can use a restart strategy during a probe to prevent STS from getting stuck in hopeless parts of the state-space. A restart strategy St is defined as $St = \{x_1, x_2, x_3, \dots\}$. A STS probe using St searches for a solution for x_1 time steps before restarting from scratch and searching for a solution for another x_2 time steps; this process goes on until the probe finishes. Luby et al. [17] present a generic restart strategy that can be used in STS's probes. In Section 2.2 we assume a restart strategy is being used. However, we did not have to use restarts in our experiments, STS always quickly found near-optimal solutions in the domains tested.

2.1 Type Systems

In general, a type system can use any information about a node to define its type. For instance, Chen suggests a general type system which counts how many children a node generates. Thus, in this case, two nodes are of the same type if they generate the same number of nodes (this is the type system we used with SS in the experiment mentioned in Section 2 in which SS did not find any solution with a three-hour time limit). In STS we include the information provided by a heuristic function in the definition of a type system. However, instead of incorporating only the heuristic value of the node s we are computing the type for, we also incorporate the heuristic value of nodes in the neighborhood of s. We use the following type system, which Lelis et al. [16] used for predicting the number of nodes expanded on an iteration of IDA*, as a base for our type systems.

> $T_c(s) = (h(s), c(s,0), \dots, c(s,H))$, where $h(s)$ is the heuristic value of node s, $c(s,k)$ is how many of s's children have heuristic value k, and H is the maximum heuristic value a node can assume.

Two nodes at a level of search will have the same T_c type if they have the same heuristic value and they generate the same number of children with the same distribution of heuristic values. We use variants of T_c in our experiments.

Beam search algorithms explore more nodes per level of search by increasing the value of B. Analogously, STS explores more nodes per level by increasing the size of the type system. Chen [2] presents a simple procedure for increasing the size of a type system T by appending a random integer to the information considered by T. For instance, one

could define $T_{c'}(s) = (T_c(s), R)$, where R is a random integer ranging from 1 to M. Thus, STS using $T_{c'}$ samples at most M different nodes for each type $u \in T_c$. Larger values of R will make STS explore more nodes during search.

2.2 Theoretical Analysis

We use the definition of an algorithm being *probabilistically approximately complete* due to Hoos and Stützle [8] to study the asymptotic behavior of STS.

DEFINITION 2. *Let $P_A(s^* \leq x)$ be the probability of search algorithm A finding a path from start state s^* to a goal state in time less than or equal to x. A is called probabilistically approximately complete if, and only if, for any solvable start state s^*, $\lim_{x\to\infty} P_A(s^* \leq x) = 1$.*

THEOREM 1. STS *with restarts is probabilistically approximately complete.*

PROOF. Every node reachable from s^* has a non-zero probability of being expanded by STS. This is because every state s of a given type u has a non-zero probability of being the representative state of u. Therefore, if the number of allowed restarts is unbounded, $\lim_{x\to\infty} P_{\texttt{STS}}(s^* \leq x) = 1$. □

Similarly, we say that STS is *probabilistically approximately optimal*, i.e., since an optimal path to a goal state has a non-zero probability of being explored, in the limit as the number of probes goes to infinity STS almost surely finds the optimal solution as long as the number of allowed restarts is unbounded.

Memory Complexity – STS keeps in memory one node for each representative pair in C and N. Thus, the maximum number of nodes stored in a frontier is $|T|$, the size of the type system employed. If d is the depth at which the solution is found, then STS keeps in memory at most $|T| \times d$ nodes.

Time Complexity – STS generates at most $|T| \times b$ nodes at a level, where b is the branching factor. Again considering that the solution is found at depth d, STS generates at most $|T| \times b \times d$ nodes.

3. THE QUALITY-SPEED TRADEOFF

WIDA* and BULB have parameters – the weight w by which WIDA* multiplies the heuristic value and the number of nodes B BULB expands at each level of search – that can be varied to trade off solution quality and speed. Analogous to this, by varying the number of probes p and the size of the type system, STS can trade solution quality for search speed. For instance, STS becomes breadth-first search when it uses a type system that assigns a different type to every node at a level of search. In this case, STS finds optimal solutions in a single probe, but it will not scale to large state spaces due to memory restrictions. The other extreme is when STS assigns every node to the same type. In this case STS becomes a random walk. By varying the size of the type system one gets a "new" search algorithm in the large spectrum of possibilities between breadth-first search and random walks. The number of probes used can also give some flexibility on the quality of the solutions found by STS and search speed. By increasing the number of probes and taking the best solution found by them, one will reduce the variance of the quality of the solution found by STS.

We illustrate the behavior of STS when varying the size of the type system and the number of probes with experiments on two different state-space representations of the blocks world domain [18]. We use two representations of the blocks world because we are also interested in studying the effect of the solution depth on the suboptimality of STS's solutions. In one representation an action corresponds to the act of moving a block from the top of a stack to the top of another stack (or to the table). In this representation the branching factor is quadratic in the number of blocks. The other representation we use has a robot hand to pick up and put down blocks. In this representation an action corresponds to the act of picking up or putting down a block, thus the branching factor is only linear on the number of blocks. Clearly, the solution length of the representation with the hand is exactly twice as long the solution length without the hand — two actions in one representation correspond to one action in the other. We call the version with the hand "deep blocks world" (DBW) and the other "shallow blocks world" (SBW).

In this experiment the results are averaged over 500 random start states with 20 blocks. These instances were solved optimally with a specialized solver by Slaney and Thiébaux [18] so that the suboptimality of STS's solutions could be computed exactly. We compute the suboptimality for a problem instance by dividing the solution cost STS found by the optimal solution cost of that problem instance, subtracting one from the result of the division and then multiplying by 100. The suboptimality we report in our tables of results is the average of the suboptimalities for the set of problem instances (denoted by Sub.). Besides the average suboptimality, we also show the average runtime (Time) in seconds. In addition to the average values we show the standard deviation of the STS results.

We use several variations of T_c. The T_c type system includes the heuristic distribution of the children of a node. In our variations of T_c we limit the number of operators to be applied to a node when computing its type. Type system T_x is the type system that allows x operators. For instance, if a node s normally generates 200 children and we are using a type system T_{50} that limits the number of operators to 50, then when computing $T_{50}(s)$ we only use the heuristic value of the 50 nodes generated by applying the first 50 operators applicable to s. Type systems that allow more operators to be applied tend to be larger than type systems that allow fewer operators as they take into account more information about the neighborhood of a node. Therefore, by varying the number of allowed operators we control the size of a T_c type system. The tradeoff is clearly observed in the experiments on the SBW shown on the left part of Table 1: as the size of the type system increases the suboptimality decreases and the runtime increases. This tradeoff is not guaranteed to occur in general, as demonstrated by the results on the DBW, shown on the right part of Table 1. Although T_c is larger than T_{18}, STS finds solutions in 0.8 seconds on average when using either type system. This fact is explained by the suboptimality of the solutions. STS expands more nodes per level when using T_c because T_c is larger than T_{18}. However, STS searches deeper when using T_{18} because STS finds longer solutions: solutions 12.8% longer than optimal when using T_{18} compared to solutions 5% longer than optimal when using T_c. The nodes expanded by STS while searching deeper in the search tree compensate for the fewer nodes expanded per level when using T_{18}.

Table 2 shows experiments when the type system is fixed (T_{140} for the SBW and T_{18} for the DBW) and we vary the

Shallow Blocks World ($p = 1$)			Deep Blocks World ($p = 1$)		
Type	Sub. (%)	Time (s.)	Type	Sub. (%)	Time (s.)
T_{100}	4.0 ± 4.6	7.93 ± 0.60	T_{16}	80.8 ± 230.6	0.79 ± 0.28
T_{120}	3.5 ± 4.0	10.22 ± 0.75	T_{17}	25.2 ± 71.6	0.76 ± 0.32
T_{140}	2.9 ± 3.4	12.56 ± 0.84	T_{18}	12.8 ± 24.0	0.80 ± 0.09
T_{160}	2.6 ± 3.4	14.12 ± 0.93	T_{19}	7.8 ± 15.4	0.83 ± 0.09
T_{180}	2.4 ± 3.3	14.91 ± 1.01	T_c	5.0 ± 3.7	0.80 ± 0.07

Table 1: STS with different type systems.

Shallow Blocks World (T_{140})			Deep Blocks World (T_{18})		
p	Sub. (%)	Time (s.)	p	Sub. (%)	Time (s.)
1	2.9 ± 3.4	12.56 ± 0.84	1	12.8 ± 24.0	0.80 ± 0.09
2	2.0 ± 2.8	25.34 ± 1.61	2	8.7 ± 20.0	1.54 ± 0.14
3	1.8 ± 2.6	38.13 ± 2.43	3	7.1 ± 12.8	2.30 ± 0.29
4	1.4 ± 2.4	50.51 ± 3.16	4	5.8 ± 9.7	3.05 ± 0.30
5	1.1 ± 2.1	63.43 ± 3.90	5	4.3 ± 5.5	3.76 ± 0.29

Table 2: STS with different number of probes.

number of probes. As p increases, the total runtime obviously must increase and the suboptimality will usually decrease. As when varying the size of a type system, it is not always true that increasing the number of probes will decrease suboptimality.

At first glance STS seems to perform better on the DBW than on the SBW. For instance, comparing T_{100} of the SBW with T_c of the DBW we note that STS produces solutions of only slightly higher suboptimality and it is almost ten times faster on the latter. However, when we increased the number of probes on the DBW so that the solutions were of equal suboptimality in both representations, we noticed that STS could be slower on the DBW. For instance, we slowly increased the number of probes for STS using T_{16} on the DBW and we noted that STS eventually became slower than STS using T_{100} on the SBW and still produced solutions of higher suboptimality. In particular, STS with 20 probes using T_{16} on the DBW produces solutions 6.6% ± 12.9% longer than optimal in 11.71s. ± 3.04s. We conjecture that STS tends to find solutions with higher suboptimality in domains with deeper solutions.

Here is an observation that supports our conjecture. STS seems to be more sensitive to variations in the size of the type system on the DBW. When using T_{16} STS finds solutions 80% longer than optimal on average, but when using the larger T_c the solutions are only 5% longer than optimal on average. This phenomenon is not observed on the results on the SBW: the average suboptimality does not change dramatically when varying the size of the type system. One possible explanation for this phenomenon is the following. Let q be the probability of STS selecting the node on the optimal path at a given level of search and d be the optimal solution length. Then, q^d is the probability of STS finding the optimal path in a probe. If q drops from 0.995 to 0.990, say, by decreasing the size of the type system, then the probability of STS finding the optimal solution on the SBW drops from 0.86 to 0.73 — a decrease of 13% —, considering solutions with 30 moves. With the same change in q, the probability of STS finding the optimal solution on the DBW drops from 0.74 to 0.54 — a decrease of 20% —, considering solutions with 60 moves. STS is more sensitive to changes in q for larger values of d. Also, obviously, for a fixed value of q STS has better chances of finding the optimal solution in domains with lower solution depths. Note that depending on the values of q and d it might take a prohibitively large number of probes to reduce the suboptimality of STS's solutions to an acceptable value.

4. EMPIRICAL COMPARISON TO OTHER HEURISTIC SEARCH ALGORITHMS

In our experiments we emulate an environment in which only weak heuristics are available. However, at the same time, we wanted to test STS in domains for which optimal solutions could be obtained, either by search algorithms using strong heuristics or by specialized solvers, so that suboptimality could be computed exactly.

The experiments described in this section are run on the 20-SBW (SBW with 20 blocks), on the 35-pancake puzzle, and on the 24-puzzle. These domains can be seen as versions of typical agent problem domains. For instance, the sliding-tile puzzle is a congested version of the pebble motion problem occurring in multi-robot motion planning; the blocks world can be seen as a warehouse problem in which a robot has to arrange the packages in a specific order before packing them into a truck so as to facilitate delivery. Taken together these three domains offer a good challenge to STS. First, the search trees in all three domains do not contain leaf nodes, thus an STS probe will only finish if reaching the goal. Second, the three domains have very distinct properties. For instance, the 20-SBW and the 35-pancake puzzle have shallow solutions: approximately 30 moves; the 24-puzzle has deep solutions: approximately 100 moves. The 35-pancake puzzle and the 20-SBW have larger branching factors: 34 for the former and between 1 (when all blocks are on a single stack) and 361 (when the blocks are spread on the table) for the latter; the 24-puzzle has a much lower branching factor: 2.36 on average [15]. We compare STS to WIDA* and BULB, and also to IDA*-BST, which is IDA* using the strong inadmissible bootstrap heuristic [10]. All our experiments were run on a 2.6 GHz machine and all four algorithms were run on the same test instances. The maximum number of nodes stored by BULB was set to 6 million nodes in all experiments.

In our experiments we chose the type system and the number of probes used by STS, the value of w used by WIDA*, and the value of B used by BULB so that the results were comparable. A result is comparable either if two algorithms are similar in one of the dimensions (suboptimality or runtime) and different in the other dimension, or if one algorithm outperforms the other in both dimensions. The procedure we used to find the values of w, B, the number of operators considered by the T_c type system, and the number of probes p was to set them initially to some value and slowly vary them until the results were comparable. We state the values of w, B, and p used in each domain in the table of results below. The description of the type systems used is given in Appendix A, together with a description of the domains and heuristic functions used. In our table of results we highlight the best results in bold. For IDA*-BST we used 5,000 bootstrap instances and used the last heuristic produced by the method after days of "batch learning" (see Jabbari Arfaee et al. [10] for details).

Table 3 presents the results. Our empirical results show that (1) on the 20-SBW and on the 35-pancake STS sub-

	35-Pancake Puzzle			20-Shallow Blocks World			24-puzzle		
Algorithm	Parameter	Sub. (%)	Time (s.)	Parameter	Sub. (%)	Time (s.)	Parameter	Sub. (%)	Time (s.)
WIDA*	w=1.4	12.7	11.42	-	-	-	w=1.9	34.2	253.30
BULB	B=5,000	16.5	34.26	B=5,000	86.0	31.79	B=12,000	**7.5**	16.02
IDA*-BST	-	15.4	21	-	9.6	23.00	-	8.1	**9.65**
STS	p=4	**8.8 ± 3.7**	**5.22 ± 0.26**	p=4	**2.4 ± 3.3**	**14.91 ± 1.01**	p=1	27.6 ± 9.2	15.58 ± 2.45

Table 3: Comparison of STS with some traditional heuristic search algorithms.

stantially outperforms WIDA*, BULB, and IDA*-BST; (2) on the 24-puzzle, STS outperforms WIDA*, but it is outperformed by BULB and IDA*-BST. WIDA* does not have entries for the 20-SBW because it was not able to solve any problem instance with a time limit of more than one day per instance (we tried the following values of w: 1.0, 1.5, 2.0, $\cdots$, 10). We do not present the number of nodes generated because they are not comparable across different algorithms. For instance, while STS generates many fewer nodes than any other algorithm in all domains tested, STS is not necessarily the fastest one, as shown by the results on the 24-puzzle. This is because the node generation in STS is more expensive than the node generation in the other algorithms because of the type computation.

Our experiments show that STS can be substantially better than other heuristic search algorithms. For instance, on the 20-SBW STS is so accurate that 70% of the test instances were solved optimally. STS is also accurate on the 35-Pancake puzzle and it is substantially better than its competitors. However, STS is outperformed on the 24-puzzle by BULB and IDA*-BST. The 24-puzzle is a domain with deep solutions (average solution depth of approximately 100). As we have conjectured, STS is more likely to find lower quality solutions in domains with large solution length. Nevertheless, STS is better than WIDA* on the 24-puzzle.

Although WIDA* performs worse than STS in all three domains and worse than BULB in two of the domains, WIDA* solves a harder problem as it finds solutions with bounded suboptimality. BULB and IDA*-BST do not have quality bounds on their solutions. STS only guarantees optimality in the limit as the time allowed for searching goes to infinity.

We have observed in experiments not shown in Table 3 that STS performs better when employing strong heuristics to define its type system. However, STS is certainly not the algorithm of choice when strong heuristics are available. For instance, in an experiment we performed on the pancake puzzle using the strong, hand-crafted GAP heuristic [7], WIDA* was always better than STS. A search algorithm should concentrate its search effort on nodes with lower heuristic value when using strong heuristics. STS ignores the fact that the heuristic might be strong and it distributes its search effort equally among nodes with different heuristic values.

5. BEAM STS

STS usually offers a good tradeoff between solution quality and search speed by varying the size of the type system and the number of probes, as discussed in Section 3. However, in some other cases, varying the number of operators available when computing a node's T_c type might not be enough to get the desirable tradeoff between solution quality and search speed. For example, one can imagine a domain in which for a fixed number of probes STS could find solutions of low quality with a T_5 type system and be too slow to be practical with a T_6 type system.

We present Beam STS (BSTS), a variant of STS that allows one to make fine adjustments on the tradeoff between solution quality and search speed. Similar to Beam Search, BSTS only expands B nodes at every level of search. However, instead of expanding the best B nodes according to the heuristic, BSTS expands the representative node of the "best" B types at every level of search; the other representative nodes are pruned. We use the heuristic to define the set of best B types. Let T be a T_c type system and n and n' be nodes in a search tree. We say that type $T(n)$ is better than a type $T(n')$ if n generates a child that has a heuristic value lower than the heuristic value of any children of n'. Intuitively, $T(n)$ is better than $T(n')$ if n generates a child that is more promising than any children generated by n' according to the heuristic function. We break ties arbitrarily when selecting the best B types at a level of search.

In contrast to STS, BSTS has the advantage that one can specify exactly the maximum number of nodes expanded at every level of search. On the downside, although BSTS still has the diversity of STS, it relies on the ranking provided by the heuristic function to select the best B types. However, our empirical results show that, due to the diversity of nodes expanded, BSTS does not suffer from the inaccuracy of the heuristic function as traditional Beam Search does. In fact, BSTS and Beam Search can also have almost opposite behaviors: all nodes expanded by Beam Search could have similar heuristic value whereas the nodes expanded by BSTS could have different heuristic values. As with STS, we do not expect BSTS to perform better than traditional heuristic search algorithms when strong heuristics are available.

We verify the effectiveness of BSTS empirically by testing it with different values of p and B on the same domains we tested STS on. We use the same heuristic functions used with STS (described in Appendix A.) We use the T_c type system for both 35-Pancake Puzzle and 20-Shallow Blocks World. For the 24-puzzle we use the T_{ggc} type system. In addition to the information that the T_c type system takes into account, when computing the type of node n, the T_{ggc} type system also accounts for the heuristic distribution and the number of grandchildren and great-grandchildren n generates. In the case of the T_{ggc} type system, we say that type $T(n)$ is better than a type $T(n')$ if n generates a great-grandchild that has a heuristic value lower than the heuristic value of any of the great-grandchildren of n'. STS would find solutions very close to optimal, but it would be too slow to be practical if using the type systems we use with BSTS in this experiment. BSTS is able to find solutions quickly while using these type systems because the number of types expanded at every level is bounded by the input parameter B.

The results of BSTS are shown in Table 4. For the values of B reported in Table 4 BSTS always quickly found a path to the goal. As in the STS experiments, we did not use a restart strategy in this experiment. We observe in Table 4 that in all three domains as we increase the number of probes p and

		35-Pancake Puzzle		20-Shallow Blocks World		24-puzzle	
p	B	Sub. (%)	Time (s.)	Sub. (%)	Time (s.)	Sub. (%)	Time (s.)
1	200	20.3 ± 6.9	1.83 ± 0.13	1.3 ± 2.3	3.15 ± 0.74	55.5 ± 30.2	1.67 ± 0.28
1	300	18.4 ± 5.6	2.53 ± 0.17	1.3 ± 2.0	4.69 ± 1.00	44.7 ± 20.9	2.63 ± 0.39
1	400	16.9 ± 5.1	3.27 ± 0.18	1.3 ± 1.9	6.28 ± 1.26	37.7 ± 14.3	3.11 ± 0.37
1	500	16.3 ± 4.7	3.81 ± 0.24	1.2 ± 2.1	7.98 ± 1.43	43.0 ± 16.4	4.14 ± 0.46
1	600	15.7 ± 6.1	4.53 ± 0.26	1.2 ± 1.9	9.39 ± 1.59	38.8 ± 14.7	4.83 ± 0.53
1	700	14.4 ± 4.5	5.10 ± 0.26	1.2 ± 2.2	10.98 ± 1.66	35.0 ± 13.2	5.52 ± 0.61
1	800	13.8 ± 4.9	5.91 ± 0.34	1.2 ± 2.0	12.63 ± 1.62	39.2 ± 21.8	6.64 ± 1.17
1	900	14.6 ± 5.5	5.80 ± 0.32	1.5 ± 2.7	14.45 ± 1.66	35.0 ± 14.5	7.58 ± 0.88
1	1000	13.3 ± 4.2	7.02 ± 0.34	1.1 ± 2.0	15.86 ± 1.65	33.2 ± 13.1	8.38 ± 0.92
5	200	12.8 ± 4.1	8.84 ± 0.45	0.4 ± 1.3	16.11 ± 3.72	29.4 ± 8.9	8.12 ± 0.79
5	300	11.3 ± 3.3	13.01 ± 0.57	0.3 ± 1.2	23.89 ± 5.05	28.8 ± 9.0	12.27 ± 1.36
5	400	10.5 ± 3.3	16.54 ± 0.58	0.6 ± 1.6	30.93 ± 6.14	24.8 ± 7.4	16.53 ± 1.56
5	500	9.9 ± 3.3	19.34 ± 0.67	0.4 ± 1.2	38.71 ± 6.96	24.5 ± 8.9	20.07 ± 2.01
5	600	9.5 ± 3.3	22.62 ± 0.79	0.4 ± 1.3	46.73 ± 7.73	24.6 ± 8.0	24.16 ± 2.29
5	700	8.9 ± 3.7	26.36 ± 0.85	0.3 ± 1.0	54.42 ± 8.06	23.4 ± 8.7	29.29 ± 2.91
5	800	8.6 ± 3.0	29.86 ± 1.56	0.4 ± 1.0	63.45 ± 8.23	24.3 ± 7.7	32.96 ± 2.93
5	900	8.8 ± 3.4	33.05 ± 1.21	0.3 ± 0.9	70.46 ± 8.27	24.6 ± 7.9	37.15 ± 3.45
5	1000	8.3 ± 3.5	34.76 ± 1.22	0.4 ± 1.1	80.71 ± 8.63	23.5 ± 8.1	40.53 ± 4.02

Table 4: Beam STS with different values of p and B.

the beam size B the suboptimality decreases and the runtime increases. The high suboptimality values for lower values of B suggest that a node n might be worth expanding even when n does not generate descendants that are considered promising according to the heuristic function. These results reinforce the principle of STS for diversifying search.

Most of the entries in Table 4 are not comparable to those in Table 3 in the sense that there is no clear winner in terms of both runtime and suboptimality. However, we observe that in a few cases for the 24-puzzle and for the 20-SBW, BSTS is better than STS. For instance, on the 24-puzzle, BSTS with $p = 5$ and $B = 400$ finds solutions of lower suboptimality quicker than STS. On the 20-SBW BSTS is better than STS for any value of B lower than 1,000 when $p = 1$.

BSTS is competitive with STS and it is substantially better than BULB (the Beam Search algorithm used in our experiments). The most impressive result is on the 20-SBW for $p = 1$ and $B = 200$: BSTS is 10 times faster and finds solutions of 66 times better quality than BULB.

The 35-Pancake Puzzle is the only domain tested in which BSTS is worse than STS. BSTS and STS expand approximately the same number of nodes per level to produce solutions 8.8% longer than optimal on the 35-Pancake Puzzle – 700 nodes per level (information not shown in the tables). The time required to compute the type of a node is reduced by reducing the number of operators available as fewer nodes are expanded per type computation. When computing a node's T_c type one has to expand 34 nodes (branching factor of the 35-Pancake Puzzle). However, if using, say, the T_5 type system, then only 5 nodes have to be expanded for each type computation. The T_5 represents a saving of 31 node expansions per type computation over T_c. Therefore, because BSTS and STS expand the same number of nodes per level and the type computation of BSTS is slower than that of STS, BSTS tends to be slower than STS.

Note that BSTS and STS could have been given the same type system in this experiment; in this case the time required to compute a node's type would be the same for both algorithms. However, that was not our goal in this experiment. We wanted to use BSTS as a way of making fine adjustments to the tradeoff between solution quality and search speed without restricting the number of operators available when computing a node's type. Our results showed that BSTS can be quite effective.

6. RELATED WORK

Korf et al. [15] also used the concept of types to efficiently predict the number of nodes expanded by IDA* for a given cost bound. Zahavi et al. [21] extended the ideas of Korf et al. and incorporated the heuristics in the type systems, which were then refined by Lelis et al. [16]. In this paper we used a type system Lelis et al. created to predict the number of nodes expanded by IDA* and applied it to the least-cost path problem with STS and BSTS.

Imai and Kishimoto [9] presented a variant of greedy best-first search (GBFS) named DBFS. DBFS uses a stochastic node expansion strategy to mitigate the inaccuracy of the heuristic being used. Instead of always expanding the most promising node according to the heuristic function, with a non-zero probability DBFS expands a random node from its frontier of nodes. GBFS and DBFS are similar to BULB in the sense that they use the ranking of nodes provided by the heuristic function to guide the search. Finally, in contrast with STS, BSTS, BULB, WIDA*, and IDA* that are memory efficient, DBFS might require a prohibitively large amount of memory to store its search frontier.

7. CONCLUSION

In this paper we presented STS and BSTS, two suboptimal heuristic search algorithms that use a different search strategy than traditional heuristic search methods. STS and BSTS use a partition of the nodes in the search tree through a type system to guide their search, which is in contrast with the estimated cost-to-go used by traditional heuristic search algorithms. Our algorithms assume that nodes of the same type will lead to solutions of the same cost. Empirical results showed that this strategy can be effective even when using weak heuristics.

8. ACKNOWLEDGEMENTS

This work was supported by the Laboratory for Computational Discovery at the University of Regina. The authors gratefully acknowledge the research support provided by Alberta Innovates - Technology Futures, AICML, and NSERC.

9. REFERENCES

[1] N. Burch and R. C. Holte. Automatic move pruning revisited. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, 2012.

[2] P.-C. Chen. *Heuristic Sampling on Backtrack Trees*. PhD thesis, Stanford University, 1989.

[3] P.-C. Chen. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, 21:295–315, 1992.

[4] J. C. Culberson and J. Schaeffer. Searching with pattern databases. In *Advances in Artificial Intelligence*, volume 1081 of *LNAI*, pages 402–416. Springer, 1996.

[5] D. Furcy and S. Koenig. Limited discrepancy beam search. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI*, pages 125–131, 2005.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[7] M. Helmert. Landmark heuristics for the pancake problem. In A. Felner and N. R. Sturtevant, editors, *Proceedings of the Third Annual Symposium on Combinatorial Search*. AAAI Press, 2010.

[8] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[9] T. Imai and A. Kishimoto. A novel technique for avoiding plateaus of greedy best-first search in satisficing planning. In *AAAI*, pages 985–991, 2011.

[10] S. Jabbari Arfaee, S. Zilles, and R. C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[11] S. Koenig and M. Likhachev. Real-time adaptive A*. In H. Nakashima, M. P. Wellman, G. Weiss, and P. Stone, editors, *AAMAS*, pages 281–288, 2006.

[12] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[13] R. E. Korf. Linear-space best-first search: Summary of results. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 588–588, 1992.

[14] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134:9–22, 2002.

[15] R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of iterative-deepening-A*. *Artificial Intelligence*, 129:199–218, 2001.

[16] L. Lelis, S. Zilles, and R. C. Holte. Improved prediction of IDA*'s performance via ϵ-truncation. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search*, 2011.

[17] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *IPL: Information Processing Letters*, 47, 1993.

[18] J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1–2):119–153, 2001.

[19] L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 756–761, Washington, DC, July 1993. AAAI Press.

[20] W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: an asynchronous branch-and-bound DCOP algorithm. In L. Padgham, D. C. Parkes, J. P. Müller, and S. Parsons, editors, *AAMAS (2)*, pages 591–598, 2008.

[21] U. Zahavi, A. Felner, N. Burch, and R. C. Holte. Predicting the performance of IDA* using conditional distributions. *Journal of Artificial Intelligence Research*, 37:41–83, 2010.

[22] U. Zahavi, A. Felner, R. C. Holte, and J. Schaeffer. Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence*, 172(4–5):514–540, 2008.

APPENDIX

A. DESCRIPTION OF THE DOMAINS, HEURISTICS AND TYPE SYSTEMS

20-SBW – The 20-SBW has more than 10^{20} reachable states. The goal state we used was the one that has all the blocks stacked on a single tower. The results were averaged over 50 random start states. We computed the suboptimality of `STS`'s solutions by optimally solving the problems with a specialized solver [18]. We used the T_{180} type system explained above in this domain. The weak heuristic function we used for `STS` in this experiment is the number of blocks out of place.

35-Pancake Puzzle – The 35-pancake puzzle has 35! reachable states. We used 50 random instances for testing. These instances were solved optimally with IDA* using the hand-crafted GAP heuristic [7]. The heuristic we used for the search algorithms in this experiment was the maximum of the dual lookup [22] of a set of seven 5-pancake pattern databases (PDBs) [4]. We used T_7 as the type system. If using the heuristic values of the 7 children separately in the type system `STS` would find paths with lower suboptimality, but it would require more time to search. Thus, the result would not be comparable to some of the entries in Table 3. Instead of using the heuristic value of each child separately as in T_c, we use the sum of the heuristic value of all 7 children to summarize the information.

24-Puzzle – The 24-puzzle has 25!/2 reachable states. We used the standard 50 instances solved optimally by Korf and Felner [14] for testing. Five 4-tile PDBs, and Manhattan Distance (MD) were used as heuristic functions in this domain. The heuristic we used was the sum of all PDBs and MD. We used the T_c type system for `STS`. In addition to the information contained in T_c, when computing the type of node s we also included the kind of location of the blank[1] of the parent of s and of s itself. For WIDA* we used the maximum of the PDBs and MD – when using the sum of the heuristics as we did for BULB and `STS`, WIDA* (or even IDA*) found very suboptimal and thus not comparable solutions.

[1]The blank can occupy one of the following kinds of positions: corner, edge, or middle